August 1, 2003

To the Graduate School:

This thesis entitled "Design and Analysis of a Performance Evaluation Standard for Parallel File Systems" and written by Frank Shorter is presented to the Graduate School of Clemson University. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Engineering.

Walter B. Ligon III, Advisor

We have reviewed this thesis
and recommend its acceptance:

Ron Sass

Adam Hoover

Accepted for the Graduate School:

Design and Analysis of a Performance Evaluation Standard for Parallel File Systems

A Thesis

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Computer Engineering

by

Frank Shorter

August 2003

Advisor: Dr. Walter B. Ligon III

ABSTRACT

Evaluating the capabilities of computer hardware has historically been a subjective and controversial subject. The following thesis will present an overview of the I/O needs of parallel programs, describe what systems are in place to meet those needs, and propose an extensible framework that gathers information from an underlying parallel filesystem to aide in the evaluation of such systems. The high performance cluster computing world has a large range of applications that have a large range of I/O needs. From scientific researchers who need to perform large out-of-core simulations, to engineers who need to store large datasets, there are many ways that I/O needs differ. Consequently, the tools used to evaluate existing systems are also quite dissimilar. In order to evaluate a parallel filesystem, it is necessary to understand what the needs of an application are. Only then can performance information be gathered and evaluated. The following text will propose a suite of access pattern benchmarks that are representative of the needs of many parallel I/O codes and attempt to reduce the degree of subjectiveness in choosing a parallel filesystem to meet the needs that parallel applications typically have.

DEDICATION

To Mom and Dad.

ACKNOWLEDGMENTS

Dr. Walt Ligon, my adviser, for all his guidance, support, and direction. Dr. Rob Ross for his guidance and feedback. Phil Carns for his everlasting patience while explaining basic things. Avery Ching and Rajeev Thakur for providing application access pattern codes. Dale Whitchurch, Curt Moore, and Larry Maner for proofreading.

TABLE OF CONTENTS

Page

TITLE PAGE i

ABSTRACT ii

DEDICATION ii

LIST OF FIGURES vi

1 Introduction 1

1.1 Parallel I/O 2
1.2 File access APIs 3
1.3 Benchmarks for parallel I/O 4
1.4 Approach 6
1.5 Thesis outline 8

2 Background and related work 10

2.1 Performance evaluation of computer resources 10
2.2 Evaluation of serial filesystems 11
2.3 Existing parallel filesystem benchmarks 12
2.3.1 b_eff_io 12
2.3.2 IOR 13
2.3.3 mpi-tile-io 14
2.3.4 Summary of past parallel benchmarks 14
2.4 Parallel file systems 15
2.5 Characterizations of workloads 18
2.5.1 The scalable I/O initiative 18
2.5.2 CHARISMA 19
2.6 I/O abstraction trends 21
2.7 Lessons learned 21

3 Design of a parallel I/O benchmark 23

3.1 High performance I/O models 23
3.1.1 Timing suite framework 24
3.2 Architecture and implementation 24
3.2.1 Timing mechanism 26
3.2.2 Timing framework organization 28
3.2.3 Configuration 28
3.3 Access pattern modules 29

3.3.1 Overview 30
3.3.2 Methods 31
3.4 Access pattern scope 35
3.4.1 Spatial access patterns 35
3.4.2 Temporal access patterns 42
3.5 Summary 45

4 Results 46

4.1 Overview 46
4.2 Access pattern test results with PVFS 46
4.2.1 Test environment 47
4.2.2 Simple strided 48
4.2.3 Nested strided 49
4.2.4 Random strided 49
4.2.5 Sequential 50
4.2.6 Segmented 51
4.2.7 Tiled I/O 53
4.2.8 Flash I/O 54
4.2.9 Unstructured mesh 55
4.3 Implementation issues 55
4.4 Summary of results 56

5 Conclusion 57

5.1 Comparability 57
5.2 Portability 58
5.3 Comprehensiveness 58
5.4 Objectivity 59
5.5 Future work 59
5.5.1 Expansion of access patterns 60
5.5.2 Framework for interpreting results 60

APPENDICES 63
A Access pattern module method parameters 63

BIBLIOGRAPHY 66

LIST OF FIGURES

Figure | Page

2.1 Noncontiguous data in memory, contiguous in file 20
2.2 Noncontiguous data in file, contiguous in memory 20
2.3 Noncontiguous data in both file and memory 20

3.1 Benchmark layout 24
3.2 Ideal way to time aggregate bandwidth 26
3.3 How we calculate aggregate bandwidth 27
3.4 Access pattern module flow 31
3.5 Simple strided access pattern 35
3.6 Nested strided access pattern 36
3.7 Random strided access pattern 37
3.8 Sequential read 38
3.9 Segmented access pattern 39
3.10 Tiled I/O 40
3.11 Flash I/O Memory Layout 41
3.12 Flash I/O Disk Layout 41

4.1 Simple strided performance 47
4.2 Nested strided performance 48
4.3 Random strided performance 49
4.4 Sequential performance 50
4.5 Sequential performance (no interleaving) 51
4.6 Segmented performance 52
4.7 Tiled performance 52
4.8 Flash performance 53
4.9 Unstructured mesh performance 55

Chapter 1

Introduction

Over the past few years, high performance computing has moved away from expensive, monolithic systems and moved toward the commodity “pile-of-pc’s” approach [1]. This technique of using many workstation class machines with a high speed interconnect has essentially lowered the barrier of entry to high performance computing. In addition to the use of lower cost hardware, the “open source revolution” [10] has contributed to the growth of high performance cluster computing. Linux and a multitude of programs released under the GNU license have provided a low cost infrastructure which has encouraged this growth.

Beowulf clusters are large groups of workstation class computers that attempt to parallelize large tasks by having each node work on a separate part of a very large problem. One node is designated as the “head node” which is where users typically launch jobs. The other nodes, which are referred to as slave nodes, are connected to the head node and the other slave nodes through a high speed network. In theory, adding more nodes to a cluster will increase the amount of work that can be done per unit time. However, there are many scalability issues that must be taken into account by those who write applications, as well as systems software programmers.

The birth of high performance cluster computing, while allowing wider access to computing resources, has also led to the need for a large amount of systems software. Since each node in a cluster has its own separate memory and disk systems, there are a number of issues involved in trying to make these areas appear as though they were one unified system. Operating systems that manage resources in the traditional Von Neumann architecture provide a number of features such as process control, memory management and disk I/O [6]. In the cluster environment, these mechanisms have developed over time. Whether these mechanisms have been addressed through formal specifications (such as message passing) [17] or through a more ad hoc process, there is a large amount of research that is ongoing in this area.

As systems software emerges to handle the resource management issues that are involved with operating a Beowulf class machine, it is necessary to have some way to evaluate their performance. Without a means to measure different implementations of systems software, vendors have no authoritative way to assert that their implementation is more effective, and systems administrators are left without a clear idea of which tools best meet their needs. Having a way to evaluate the performance of systems software is important in all areas of parallel computing.

1.1 Parallel I/O

One of the important areas of research involving cluster based systems is the filesystem. Instead of a many small segmented storage regions, it is more convenient for users to access the aggregate disk space of all nodes in a cluster through one global namespace. The key idea here would be that the underlying file system has to somehow split up data and figure out how to store it on some set of nodes such that any other node can access it, and that these accesses are speedy. This process of splitting up a file and distributing it across multiple nodes in a cluster is known as “de-clustering.”

There are a number of parallel filesystems which provide this global namespace and store data on multiple nodes. Some filesystems aim to provide high performance, while others aim to provide fault tolerance and high availability. While both approaches solve the unified namespace issue, there is still the performance issue that needs to be addressed.

Evaluating performance of a parallel filesystem is not a straightforward task. Determining how to time events in a distributed memory machine can be troublesome due to the fact that each node may not share the same clock. Varying propagation delays (due to topology) and heavy utilization may cause barrier synchronizations to take a non-uniform amount of time to complete. In addition to figuring out how to time events, it must be decided what events should be timed in order to get a realistic view of the performance of the system on the whole.

That being said, the only way to get an accurate idea of the I/O performance of any system would be to run actual application code on a given system. Synthetic benchmarks can provide some knowledge about overall performance, but obviously, running the target application code is the only way to know exactly how well a system will perform.

1.2 File access APIs

In order to access any file on a parallel filesystem, it is necessary to have some set of functions to manipulate files. When dealing with traditional block device filesystems, the POSIX interface provides a set of functions for input, output, and metadata manipulation [11]. While parallel I/O can be achieved through creative use of the POSIX interface, there are file access APIs designed specifically for parallel I/O.

MPI-IO

One such interface is the MPI-IO standard that was described as part of the MPI-2 specification [7, 18]. MPI-IO aims to provide a layer of abstraction for I/O and to ensure portability across multiple architectures and filesystems. Like POSIX, MPI-IO supports the idea of file handles, and seeking. However, MPI-IO also introduces several new concepts related to file access. MPI-IO provides the concept of a "File view" where the caller can specify a virtual "view" of a file, and transparently access some set of bytes in a file without explicitly moving the file pointer. MPI-IO can provide behind the scene optimizations to coalesce smaller accesses into large contiguous reads or writes. One such optimization is known as data sieving [32]. It is also possible to provide "hints" to the MPI-IO implementation in order to specify a communication strategy. If the MPI-IO implementation has knowledge about the type of access beforehand, it may be able to reduce the number of intermediate buffers it uses.

Essentially, MPI-IO provides a portable set of file access functions that provide many capabilities to efficiently access files on parallel filesystems on a variety of systems.

Other parallel-I/O APIs

In addition to the MPI-IO specification, there are other parallel I/O standards that are used by scientific and engineering applications. While there are a number of parallel I/O APIs for manipulating data, many of them are built on top of MPI-IO. Since these APIs call MPI-IO functions, any performance slowdown would be introduced by that API and not the filesystem. These APIs are a valuable resource to see how parallel codes interact with files. Instead of viewing files as a single linear array of bytes, they provide alternate descriptions of files. Namely, they can model multi-dimensional datasets, and provide object view capability.

1.3 Benchmarks for parallel I/O

The goal of benchmarking a parallel filesystem is to gain a complete idea of the I/O capabilities of a system. As such, a benchmark should imitate the needs of a set of programs for which the system is to be used. It is necessary to get an idea of the common I/O needs of a parallel application, and how they are typically met on a distributed memory machine. How do the applications interact with files? Are they writing large buffers or small ones. Are they accessing non-contiguous regions on disk? Is the application accessing non-contiguous regions in memory? What is the stride distance in the file between consecutive accesses? What type of access pattern does a given application use? These are only some of the questions that must be addressed in the design of a parallel I/O benchmark. These access pattern descriptions are key. Spatial access patterns as well as temporal access patterns will be examined.

While there are a number of MPI-IO benchmarks to test parallel I/O, they typically have dissimilar reporting mechanisms, as well as dissimilar timing mechanisms. Many of the MPI-IO benchmark codes also do not get enough coverage due to a limited selection of access patterns. While it is not necessary to implement a benchmark that covers the entire set of possible access patterns, it is necessary to provide enough different patterns to ensure that an accurate picture of the I/O capabilities of a parallel filesystem is returned. In addition to a representative collection of access patterns, standardized timing and reporting mechanisms must also be used. Results that are not standardized and not directly comparable to each other are obviously not helpful in evaluating I/O performance.

After a large selection of results has been gathered, it is necessary to have a framework to interpret these results. Any tool that performs interpretation will have to exercise care in weighting results. That is why such a framework for interpretation should merely highlight results from the slew of results that will be returned. This type of weighting will ultimately depend on the overall I/O needs of the specific system being evaluated. For example, if the system under evaluation testing needs to be used to store large datasets for an out-of-core application, the access patterns that best describes that particular simulation would need to receive the most attention.

1.4 Approach

With a range of possible parallel filesystems and I/O needs, it is necessary to have some way to evaluate performance. A benchmark should provide a mechanism to gather information about the I/O capabilities of a system, and be able to compare these results. While there are some existing parallel benchmarks, they do not provide enough coverage in terms of commonly used access patterns. This document will propose standard tests and procedures for evaluating parallel I/O performance, provide a tool that automates workload simulation with a variety of access patterns, and examine the issues inherent in evaluating I/O performance. Our thesis is that the proposed standards and tool will provide a comparable, portable, comprehensive, and objective evaluation of a parallel filesystem. The standards that this benchmarking tool will be measured against are described in detail as follows:

- **Comparability**: It is key that the results taken from one system can be compared to other systems. While not all access pattern variations are comparable to each other, it is possible to compare performance of a certain access pattern on one system to its performance on a different system. Comparability not only denotes the ability to compare two distinct parallel filesystems, but to compare configuration options and optimizations within a single parallel file system. For example, system administrators would be able to use such a tool to explore the effects of changing the number of I/O servers, striping sizes, or caching mechanisms. The use of standardized reporting and timing will aid in the comparability.

- **Portability**: While the target of this benchmark is distributed memory Linux clusters, it should be possible to run this tool on any machine that has a C compiler and supports message passing. The use of these two standards are very important in maintaining portability. By portability, we also intend for this evaluation tool to be able to operate in a heterogeneous environment. Portable standards and interfaces will be key in facilitating this.

- **Comprehensiveness**: A multitude of access patterns will be available to ensure that a maximum amount of information is available about the performance of a filesystem. While it may not be possible to implement every widely used access pattern immediately, the framework that we have established will allow for rapid development of new access patterns as application programmers describe them. Standardized reporting mechanisms are one area that parallel evaluation tools have been lacking in the past. In addition to simply displaying the throughput rates for a given transfer size, it would be beneficial to have some way to highlight the access patterns that are more relevant to a particular application. In addition to having a large collection of access patterns, it should be easy to add new access patterns to this system. Modularity is important as a means of achieving a large amount of comprehensive results. It should be relatively simple to add an access pattern given a high level understanding of how the file access is occurring. Presenting a cohesive model for I/O manipulation will be important to rapidly adding new access patterns.

- **Objectivity**: This tool will provide a mechanism for the end user to independently verify the vendor claims about the performance of a filesystem. Being able to manipulate filesystem parameters to gauge performance trends is important as is any weighting done to judge the performance of one system relative to another. This tool will be released to the community with the source under the GNU license should independent parties wish to verify any weighting algorithms.

We hope that this standard will be a significant contribution to the community. The implementation of this standard is necessary for the following reasons:

- **Existing access pattern coverage in benchmarks is insufficient**: While there are a number of parallel I/O benchmarks, there needs to be more access pattern coverage such that the most possible amount of information can be retrieved.

- **Vendors claims need independent verification**: While it is useful for vendors to describe the capability of their system software, there should be a tool available to the public that allows these tests to be repeated by a client, on their system at their convenience.

- **I/O system tuning is dependent on the client's needs**: Depending on the application that must run on any given system, there will be options that can be tuned to cause that application performance to vary. The optimal settings may not be immediately apparent, and this tool aims to help make subtle changes more obvious.

The tool that will be developed to meet these requirements will be able to gather a wide range of results from a number of access pattern workloads. While performance data can be obtained, there is still a large amount of interpretation that will need to be done to obtain useful information about the performance at each data point. For example, it may be possible to reduce the amount of data returned to the user to a simplified ranking. Similarly, it may be possible to reduce the large amount of information that is gathered to a single ranking, or classification across a category of workloads. This type of analysis of performance data opens many questions and is will be left open for future works to explore. Further suggestions for possible directions of analysis will be presented in the future works section.

1.5 Thesis outline

In chapter two, this document will present an overview of the past approaches taken to benchmark parallel I/O systems, present studies that have analyzed application needs from a filesystem perspective, present a collection parallel filesystems that introduced new ideas, and discuss parallel I/O abstraction trends. Chapter three will present the design of this benchmarking standard; namely, how the design is modular and what common access patterns will be studied. Chapter four will present the results of each access pattern on a testbed running PVFS, and discuss the effectiveness of each access pattern, against the known characteristics of PVFS. Chapter five will examine how the results of the analyzation matched up with the goals set forth in section 1.4 as well as presenting ideas for future exploration of this subject.

Chapter 2

Background and related work

2.1 Performance evaluation of computer resources

Being able to characterize the speeds of various computer systems critically and objectively is the basis of performance evaluation. Rating computational abilities proves to be useful for both users and vendors of computer systems. However, benchmarking has traditionally been troublesome for several reasons. Correct behavior, timing/gathering data, and workload simulation accuracy are several areas that have spawned disagreement.

Much conjecture exists due to the difficulty in obtaining accurate data for a synthetic, portable benchmark. There have been many benchmarking efforts that have attempted to classify general performance of computing systems. Some of the well known general efforts are SPEC [31], Linpack [13], and the NAS Parallel Benchmarks [21]. Each of these benchmarks has a similar goal in mind: to gather data from a computing system because "...an ounce of honest data [is] worth more than a pound of marketing hype" [31].

The approaches of these benchmarks differs widely depending on what they are measuring. Some systems are a sequence of multiple micro-benchmarks [21, 16, 31], while others are one large monolithic benchmark that incorporate multiple facets of a single problem [13]. In addition to incorporating multiple micro-benchmark tests, many benchmark suites attempt to ensure that the functionality they are timing is something an application would commonly do. Since all of these synthetic benchmarks simulate specific system loads it is important that they do things that real world applications commonly do. While these benchmarking efforts may not directly stress I/O subsystems, the approach they take toward timing and reporting strategies will be useful in designing a parallel I/O benchmark.

2.2 Evaluation of serial filesystems

When evaluating I/O performance of a serial filesystem, the characteristics of this performance usually resembles that of the underlying physical drive on which the filesystem resides [25]. Access times and bandwidth vary due to the specifications of the physical drive mechanism. Sequential access is typically quite fast due to the fact that the hard drive head moves in a sequential manner over the platters of the physical disk. This type of serial access is fast because that is how the drive retrieves bytes from the physical device. This has resulted in a number of I/O benchmarks that essentially test physical disk performance instead of filesystem performance [25]. Serial disk benchmarks are good tools for determining the performance of serial filesystems; however, these benchmarks tend to ignore concurrent file access altogether. The plethora of serial filesystem benchmarks do not take into account the ways that file access must be different on a parallel machine in order to efficiently access data in parallel.

It is in this respect that serial filesystem benchmarking tools are inadequate and unable to simulate a parallel workload which a typical parallel application may provide. Therefore, any tool that simulates parallel I/O must take into account the methodology that parallel applications use to access files. Determining a "typical application" and a "typical workload" is not an easy task. Since there are some similarities in how parallel codes interact with files, it is possible to classify load in terms of access patterns. These access patterns can help describe which regions of a file certain processes wish to access. Each access will be a sequence of reads or writes.

In addition to actual I/O, There is a metadata component that serial filesystems explore. Creating directories, retrieving statistics about a file, truncating files, and deleting files are all things that a parallel application may do. As such, these actions are also candidates to be timed.

2.3 Existing parallel filesystem benchmarks

There are several existing parallel benchmarks that profile I/O performance on distributed memory machines. Some benchmarks incorporate a wide range of access patterns, while others simulate specific workloads. The following sections will introduce a few prominent existing benchmarks.

2.3.1 b_eff_io

The "Effective I/O Bandwidth Benchmark" (b_eff_io) brings together several file access patterns and examines the time that it takes to transfer data from a location in memory, to a location in a file. This benchmark classifies the parameters that a parallel benchmark may encounter into six distinct groups: application parameters, usage aspects, programming interface, access methods, filesystem parameters, and statistical aspects [25]. According to the b_eff_io benchmark, application parameters can be classified by things such as the way data is organized into memory (contiguous/non-contiguous), the way data is to be written to or read from file (contiguous/non-contiguous). Size of memory pages, size of disk blocks, and the distribution of those blocks also fall into this category. Usage aspects involve the number of processes used, as well as if a process is multi-threaded. The programming interface aspect involves the choice of file access API. The b_eff_io benchmark uses MPI-IO "...because it should be a portable interface of an optimal implementation on top of

POSIX I/O or the special filesystem I/O." [25]. The b_eff_io paper continues by stating that the choice of MPI-IO as a file access API also brought up many peripheral file access issues such as whether to use explicit offsets versus implicit offsets. Also, file locking and asynchronous I/O were areas that were examined while they implemented their code. The filesystem parameters classification that they define covers things such as the the number of I/O servers, how much data is stored in a striping unit on a server, and the physical disk block size that the I/O server uses. This benchmark also explores the issues involved in gathering meaningful data based on statistical normalization and how multiple test runs should be averaged together. The b_eff_io benchmark goes on to state that they are providing a snapshot of overall performance based upon the "coffee cup rule" (the benchmark's execution time should be relatively short; roughly the same amount of time to brew a cup of coffee) [12]. They sample many important patterns, but only run enough of each pattern to record the time necessary to transfer the entire contents of system memory to a file. Many of the ideas about standardization through MPI-IO and access pattern descriptions presented in this benchmarking effort are significant due to their novelty. However, adding new access patterns to their framework was found to be troublesome by this author.

2.3.2 IOR

The next parallel I/O benchmark that will be examined is called IOR, which stands for "Interleaved or Random." It was developed at Lawrence Livermore National Laboratory and is available in POSIX and MPI-IO flavors. IOR reads and writes contiguous buffers to non-overlapping areas of a file [34]. At the time of this writing, the IOR web page states that all random access has been removed from the IOR benchmark. The control flow of IOR is as follows: a test file is created, data is written by each process at some offset into the file, the data is then read back by a different process, the file is deleted, and throughput information is returned. IOR provides the ability to repeat a given set any number of times, as well as vary a number of block sizes via environment variables. IOR is different from b_eff_io in that it only tests one access pattern while b_eff_io tests a range of access patterns. Also, b_eff_io tests a tiny amount of the access pattern, while IOR reads or writes potentially large files, and this may take a long time to complete. Additionally, IOR offers both POSIX and MPI-IO implementations. POSIX views files primarily as a stream of bytes, while MPI-IO offers mechanism that allow regions of files to be represented as a datatype transparently. Even though it is possible to provide a workload that corresponds to a given access pattern by both methods, maintaining two versions of a benchmark that essentially do the same thing seems repetitive.

2.3.3 mpi-tile-io

In addition to b_eff_io and IOR, mpi-tile-io is a benchmark that is stresses the I/O system of distributed memory machines to simulate the needs of a real application. The mpi-tile-io benchmark provides a workload where each process accesses a "virtual tile." This benchmark simulates the workload that large visualization programs provide when they need to break up one large scene across multiple monitors (like in a video wall). Each process is assigned a tile region to access, and accesses contiguous data in a sequential manner. Obviously, the name of mpi-tile-io indicates that the file access API in use is MPI-IO, and that it reads or writes tiles that are logically distributed throughout a file. This benchmark does not explicitly dole out regions of access for each process, but lets MPI choose the tile region which is most efficient for a given process to access [20]. This benchmark heavily relies upon the use of complex MPI datatypes to accomplish its file access.

2.3.4 Summary of past parallel benchmarks

Each of these benchmarks present new ideas that showcase a number of benchmark design decisions. Using these benchmarks as an example, it would be advantageous to understand why certain design decisions were made, so that one can come away with an understanding of how parallel workloads are typically measured. Common themes that are important would be the reliance upon a standard API to perform I/O, the idea of a pattern of access, and the repeated element of work that occurs multiple times and should be timed. However, each of these benchmarks approaches timing differently. Timing events in a distributed memory machine is troublesome due to propagation delay between nodes, as well as the fact that each node uses an independent clock. Since these separate physical clocks may drift or become inaccurate over time, timing is a component that needs to be addressed carefully. The fact that each of these benchmarks handles timing differently is noteworthy given that its such an integral part of benchmarking.

2.4 Parallel file systems

There are a number of parallel file systems that exist. This section will give a high level overview of several parallel filesystems. Each of the following parallel filesystems has presented new ways of solving the problems that parallel I/O presents:

- **Vesta**: The Vesta filesystem uses subfiles to achieve parallelism. For each file that exists, there are a number of subfiles that describe portions of that file. In the Vesta world, metadata for a file describes the mapping of the physical data contained in a subfile to its corresponding location within the logical file that it comprises. [2]. Vesta attempts to break away from the notion of physical blocks and refers to one or more physical blocks of a file as a "cell." At the application level, Vesta also offers the ability to view a file in a two-dimensional view. This is accomplished by manipulation of the subfiles.

- **Galley**: While the Galley filesystem also operates with the idea of subfiles, each subfile is managed at a higher level by an object called a fork. Forks provide an abstraction that Galley uses to aid in the management of subfiles. For example, it would be advantageous to group certain regions of files together depending on the type of access pattern that will be used to read or write data. This grouping mechanism optimizes accesses as requests arrive [23]. The Galley filesystem relies upon the client/server model where each process is designated as an I/O processor or a compute processor. The compute processors are clients and the I/O processors are servers. The compute processors submit I/O requests to the I/O processors as needed. The compute process interacts with the subfiles and forks directly.

- **Lustre**: The Lustre filesystem consists of Object Storage Targets(OST), Metadata Servers (MDSs), and the Lustre Client File System. Each Lustre filesystem runs 2 MDSs: a primary and a fail-over. The MDS component handles metadata and locking, while the OST component handles actual file I/O. The Lustre Client which must be loaded on each node that is performing access will contact either the MDS to perform metadata operations, or contact a set of OSTs. The client does this through an abstraction layer called the Logical Object Volume Driver(LOV). The LOV driver maintains fault tolerance in I/O operations by switching over to an OST with replicated segments of a file [14].

- **xFS**: As part of the "Network Of Workstations" (NOW) project at The University of California at Berkley, several researchers designed a distributed, parallel filesystem called xFS [15]. The xFS paper [36] describes a distributed filesystem that is targeted for use on Wide Area Networks. Since network topology dictates where servers must run due to their bandwidth resources, this filesystem has a good deal of caching implemented to save time on data transfers. Data is only written "on demand" [36] which results in less bytes transfered when there is only one writer, but can result in slower throughput in false sharing cases [15]. While xFS is more of a distributed filesystem than a parallel one, the ideas that are presented with regards to client side caching are novel and interesting.

- **GPFS**: This filesystem is the enterprise cluster filesystem written by IBM for distributed memory Linux clusters [29]. Originally designed to run on the IBM SP [3], this filesystem has been ported to distributed memory clusters of workstations. GPFS achieves parallelism and high throughput by striping file segments across multiple I/O servers [9].

- **The Parallel Virtual File System**: PVFS consists of 3 major components: the I/O daemon (IOD), the Manager daemon (MGR), and a client side access library. The MGR provides the metadata component of the filesystem. It is the ultimate authority for everything that has to do with permissions, attributes, or directory/path information. Strips of a file are stored on each IOD according to a striping parameter that is provided at the time the file is created. The size of these strips is defined at compile time. The IODs service I/O requests and provide any sections of the file that may reside within regions that they hold. On the client side, depending on whether an application is accessing a PVFS file via the optional kernel interface, or through the PVFS client library, the PVFSD client side daemon may or may not be used. The optional kernel interface uses the PVFSD client side interface, but it is possible to access PVFS files directly without incurring the overhead of going through the kernel [5]. PVFS stripes file segments across each IOD in the system in a round-robin distribution. Striping parameters are defined on a per file basis at creation time and may not be changed. In addition to explicit control over striping, PVFS offers a mechanism to access files in a simple strided access pattern via the client library [28]. This feature allows a client application to make one request to read non-contiguous areas of a file.

2.5 Characterizations of workloads

Now that several parallel filesystems have been introduced, we must understand the workload that they will be asked to bear. There have been number of studies that attempt to characterize the workload of parallel applications, and this section will highlight a few of them.

2.5.1 The scalable I/O initiative

The scalable I/O initiative was an early attempt to characterize the I/O needs of parallel applications. According to their publications [4, 8, 26], they sampled applications that performed simulations in Biology, Chemistry, Earth Sciences, Engineering, Graphics, and Physics [8]. According to Reed et al. [26] the source code of many applications was analyzed and profiling information was gathered with the use of the Pablo performance analysis environment. Through the use of the Pablo trace library, information was gathered about what requests were made as well as the sizes of those requests. Although much information was gathered, it is stated in [4] that there was too much variation in the spatial and temporal access patterns as well as the access sizes by each of these applications to come away with useful optimization information. However, it was interesting how they distinguished between spatial access patterns and temporal access patterns. This differentiation between the relationship of the physical mapping of readers and writers to various physical locations in the file, and the relationship between accesses that are separated by time is interesting due to the cyclic behavior of the spatial access pattern, and the intermittent nature of the temporal access pattern. Essentially, they concluded that the applications they studied made regularly spaced accesses to specific locations in a file, but that those requests were made in an intermittent and non-regular manner with regards to time. Due to this irregular nature of the IO requests, they suggest a reactive mechanism to handle prefetching and cache data in order to improve performance [4].

2.5.2 CHARISMA

The CHARISMA project was another attempt to characterize the workloads of a parallel I/O systems. As part of the project, Nieuwejaar, Kotz, et al. recorded specific data relating to the number, size, and types of I/O requests for various scientific applications on the CM-5 and Intel iPSC/860. They divided all the applications that were tested into 3 divisions: General workload, Scientific vector applications, and Scientific parallel applications [24]. Their work breaks down some of the common access patterns that many scientific applications used to interact with the filesystem. The first access pattern that they recognized was dubbed “Simple strided” [24]. They go on to state that many of the applications they studied actually used sequential access, which can be mathematically decomposed to the simple strided access pattern. With a stride distance of the strip size, a simple strided access pattern essentially becomes sequential access (see figure 3.5 in section 3.4). Secondly, they go on to define “Nested strided” which may have any number of simple strides as the internal access inside a stripe. In this earlier paper [22] Nieuwejaar and Kotz differentiate between “sequential” and “consecutive” requests. Sequential requests are defined as a request “...that is at a higher file offset than the previous request from the same compute node.” Consecutive requests are defined as “...a sequential request that begins where the previous request ended” [22]. This definition of sequential is broad enough to encompass many types of strided access. In fact, they go on to claim that upward of 80% of their accesses were the result of a type of strided access [22]. The large number of strided accesses led to a need for a distinction to be drawn between “simple strided” and “nested strided.” Please see figure 3.5 for a diagram of simple strided and see figure 3.6 for a diagram of a nested strided access [24].

Figure 2.1: Noncontiguous data in memory, contiguous in file

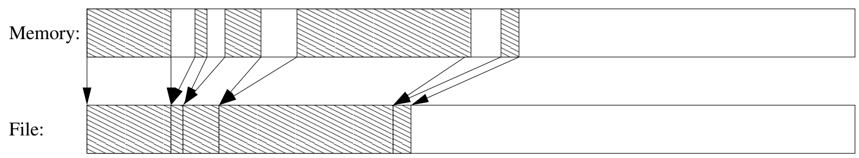


Figure 2.2: Noncontiguous data in file, contiguous in memory

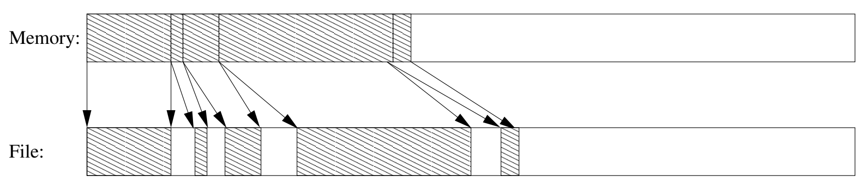
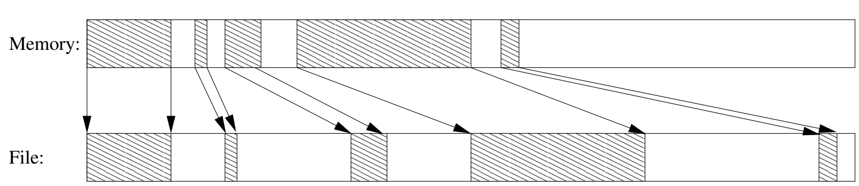


Figure 2.3: Noncontiguous data in both file and memory



2.6 I/O abstraction trends

One of the important features that MPI-IO provides is the ability to set a virtual file view. This feature allows a program to reduce the number of requests that it makes and transparently access areas of the file that may not be contiguous [19]. Instead of explicitly seeking to an offset into a file and accessing a certain number of bytes, a file view can implicitly retrieve the same data with one read or write call at the MPI-IO level [33]. MPI datatypes can be built upon one another, such that complex structures of any size and type can be defined. This means that one can specify virtually any access pattern via a file view and a derived MPI datatype. As seen in figures 2.1, 2.2, and 2.3, It is possible for data to be contiguous in file, and non-contiguous in memory; non-contiguous in file, and contiguous in memory; contiguous in both; or non-contiguous in both. With MPI derived datatypes, it is possible to describe describe access patterns within a file and within memory such that the the application only needs to submit a single request to execute the I/O transfer.

2.7 Lessons learned

Given the history of performance evaluation in other areas of computational resources, the history of performance evaluation in serial filesystems, existing parallel benchmarks, and parallel workload studies, we are left with a several possible approaches to evaluating parallel filesystem performance. Benchmarking other areas of computer resources has shown that any synthetic benchmark should mimic application performance as well as critically timing this access. Furthermore, the issue of timing "actual performance" is not an easy thing to classify due to the effects of caching and consistency. The effects of caching can easily skew the results of a benchmark, but cached access can provide a real performance boost in speed for an application. If caching data can be timed realistically with regards to how it affects application performance, there is no reason why these effects should not be included as a legitimate performance boost. What are things that a typical parallel application does? Prior workload studies have indicated that they do a variety of things, but that striped access patterns (both simple and nested) are important. In addition to spatial layout within a file, parallel applications also seem to vary their accesses over time, so temporal access patterns may make it difficult for a filesystem to cache or prefetch data. It is important for a benchmark to test a variety of access patterns that are widely used and showcase typical filesystem performance. Since it is possible to rapidly represent a number of data layouts in memory and file using MPI derived datatypes, it is possible to develop a model for I/O that can be easily timed. After defining the region of access through an MPI datatype, all that remains is the actual call to perform access (a read or a write). Figuring out a way to time this access within a distributed memory cluster machine is also extremely important. It is in this regard that we can learn from previous benchmarking efforts, namely the Linpack initiatives. Linpack times the number of floating point operations that occur over a period of time. Since these operations are significantly more finely grained than file access, it would be advantageous to examine the Linpack timing code. All of these factors will influence the decisions that will be made in designing a parallel I/O benchmark.

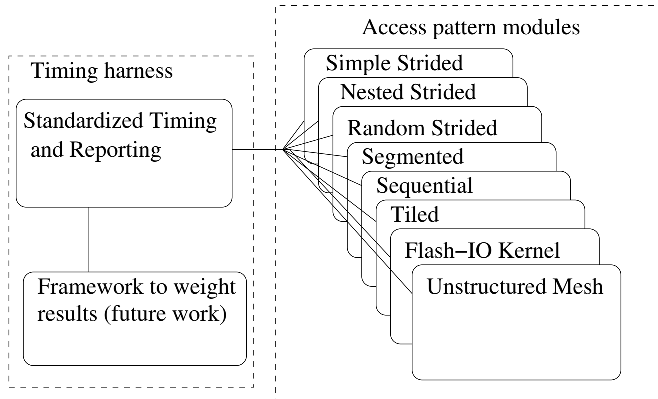
Chapter 3

Design of a parallel I/O benchmark

3.1 High performance I/O models

As shown by multiple workload studies [4, 8, 26, 24] parallel applications vary wildly in the access patterns they use to interact with files. Even though there is much variation in terms of the pattern of bytes that is accessed from application to application, these studies have shown is that there are systematic patterns by which each parallel applications will access data as they run. In general, each application will open a file, access data via a number of requests, and close the file. This idea of repeated accesses can be generalized into model that will allow us to isolate these I/O requests from the computational aspect of a parallel application. To do this, we will need to develop an abstraction that will allow us to exclusively time the I/O calls. After these I/O requests have been timed, the results will be presented so that performance can be evaluated. It is in the following chapter that an extensible framework for simulating access patterns is proposed, how timing is accomplished, and how results are to be reported.

Figure 3.1: Benchmark layout



3.1.1 Timing suite framework

To develop a modular framework requires the identification of certain key repeated elements that we can combine into similar functions. It is necessary to test a number of access patterns in a manner that allows for consistent timing and reporting. In each access pattern test, there exists a section where setup must occur, some set of operations must be timed, and cleanup must be performed. These three operations can be isolated in a way that makes it possible to test a large number of access patterns while maintaining uniform timing and reporting standards.

As shown in Figure 3.1, this timing framework will need to identify key elements which exist in multiple access patterns, and abstract those elements to a higher level where a critical unit of work can be timed.

3.2 Architecture and implementation

For the purposes of this benchmark, we are going to generalize file access into the following three phases: initialization/datatype step, a sequence of reads or writes (which we will refer to as a "work unit"), and datatype deallocation/cleanup. This model assumes that the data to be read or written has already been setup by the time work units are executed. This benchmark is concerned with the transmission of data to its intended location, not the generation of such data. The following actions are common things that an access pattern benchmark will have to do to the file under test:

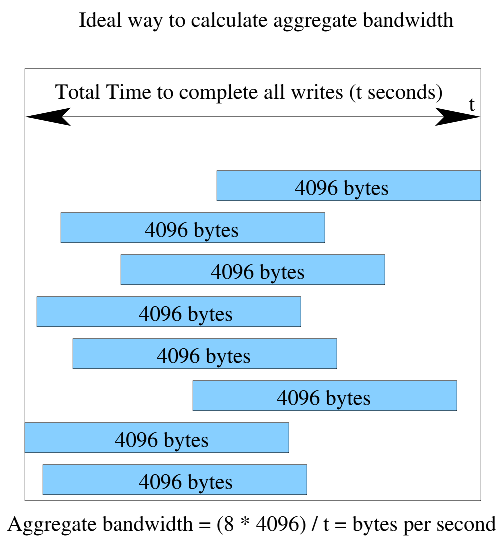
- **initialization/setup**: It is during the initialization that the file is opened, and any datatypes that are needed to describe memory or file regions of access should be created (see section 2.6 for a thorough discussion of the features MPI-IO provides for interacting with files). Since the goal of benchmarking is evaluating performance, less emphasis will be placed on verifying correctness and consistency of an underlying filesystem.

- **work units**: It is necessary to come away with a generalized way to describe the repeated I/O calls that many applications make, so in absence of a better term, this author will refer to those as "work units." A work unit is comprised of the amount of data that it accesses, and the locations of the data within the file. An access pattern module must define the regions that it intends to access, and this definition will be in terms of MPI datatypes. It is possible to mathematically describe work units for many access patterns, and this will be key factor that allows the implementation of a variety of access patterns. The work unit is the most costly phase of I/O, and will be timed.

- **cleanup**: During the cleanup phase, the file under test should be closed, any buffers should be deallocated, and any MPI datatypes should be uncommitted from the system.

The Parallel I/O benchmarking (referred to as "pio-bench" from this point forward) suite that has been implemented as part of this evaluation standard uses MPI-IO as the mechanism by which it accesses files. Since MPI-IO is the standard for parallel I/O it makes more sense to write a benchmark that uses MPI-IO to ensure portability.

Figure 3.2: Ideal way to time aggregate bandwidth



3.2.1 Timing mechanism

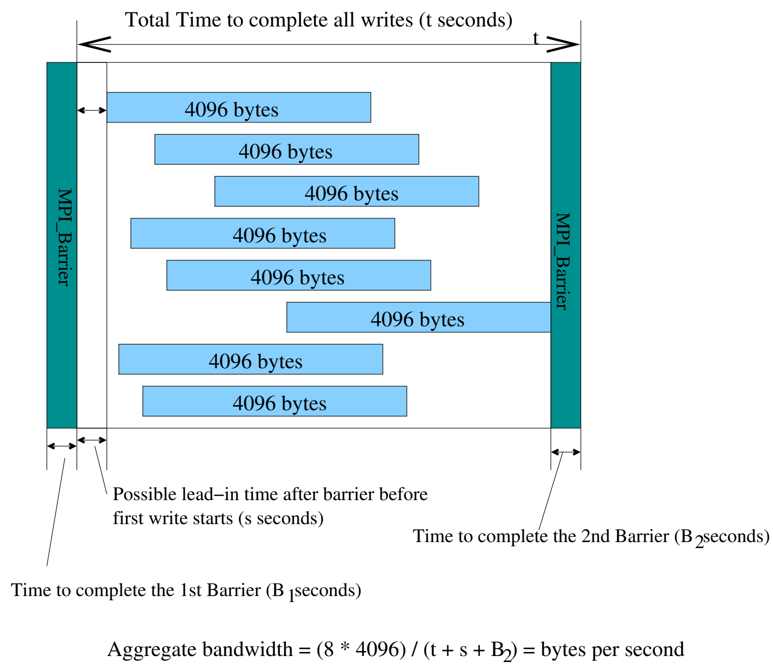
Access patterns modules contain several elements that should be timed; however, timing events that occur on multiple remote hosts in a distributed memory cluster machine is not trivial.

In figure 3.2, the ideal way to calculate aggregate bandwidth is shown. Timing would begin at the very beginning of the first access, and end when the last access finishes. However, synchronizing multiple distinct physical clocks is a complex problem in high performance computing that has not been completely solved yet. The best that can be done is to time events as follows:

$$\text{aggregate time } = T + S + B_2$$

As shown in Figure 3.3 B_1 represents the amount of time that a preliminary barrier takes (which is not included in the calculation), S represents any startup time between the end of the first barrier and the beginning of the first access on any process, T represents the

Figure 3.3: How we calculate aggregate bandwidth



total amount of time that all the accesses take from beginning to end, and B_2 represents the amount of time that a second barrier takes after each access has completed.

Since it is difficult to come up with a general case to time events that are occurring under the watch of separate physical clocks, including the second barrier time is the only way to known that all processes have finished executing a unit of file access.

The root process (process 0) essentially elects itself to act as the authority to determine when all other processes have finished. The inclusion of a barrier synchronization is necessary to ensure that the operation has completed on all nodes, and provides us with an upper bound on the aggregate execution time of a given work unit. While the aggregate time may be lower without the barrier synchronization, we are ensuring that we will not falsely report a faster time. This way, the fastest time we report includes the time after a barrier synchronization, and all processes have finished their work unit.

Directive name	Description of configuration directive
TestFile	File name of the unit under test.
ShowAverages	Instead of showing the results from each repetition of a module, this option averages the results from each buffer size for each test in each module.
ReportOpenTime	Toggles the display of the amount of time it takes to open the file under test.
ReportCloseTime	Toggles the display of the amount of time it takes to close the file under test.
ReportHeaderOffset	Toggles the display of the amount of bytes each access pattern skips at the beginning of a file before it begins accessing data.
SyncWrites	Toggles whether each write should be flushed to disk.

Table 3.1: Global timing framework configuration options.

Directive name	Description of configuration directive
<ap_module>, *</ap_module>*	These signal the beginning and end of a module block. The 3 following directives apply on a per module basis.
ModuleName	The name of a module as specified in the *PIOB_access_table* configuration field at compile time.
ModuleReps	How many times should an access pattern module run.
ModuleSettleTime	Time in seconds that the harness should wait after executing an access pattern module.

Table 3.2: Access pattern module directives.

3.2.2 Timing framework organization

The access pattern modules are organized as shown in figure 3.1. The pio-bench timing framework handles all of the timing and reporting functionality, so the access patterns only have to deal with actual I/O calls. Our generalized model of the phases of file I/O consisted of three methods to achieve virtually any pattern of access with the help of MPI derived datatypes and file views. Section 3.3.2 will detail the implementation of the access pattern modules.

3.2.3 Configuration

Since access patterns may need many different types of pieces of information to describe what type of access they should perform, deciding on a dynamic run-time configuration

Example:	
TestFile	"/mnt/pvfs/file_under_test"
<ap_module>	
	ModuleName "Simple Strided (write)"
	ModuleReps 100
	ModuleSettleTime 30
</ap_module>	
<ap_module>	
	ModuleName "Nested Strided (re-write)"
	ModuleReps 50
	ModuleSettleTime 10
</ap_module>	

Table 3.3: Global timing harness configuration example.

mechanism was not an easy task. Existing benchmarks use a variety of methods to accomplish run-time parameter configurations. Some benchmarks use command line arguments to specify how many repetitions of a test should be performed, while other benchmarks use environmental variables to achieve this. This benchmark uses apache-style configuration files using the Dotconf parser for the timing framework configuration, and the specific access pattern module configuration options. There are some run time options that do not need to change from access pattern module to another. These global options are shown in table 3.1. In addition to the global directives, there are a number of directives controlling which access pattern modules should be executed. These directives can be found in table 3.2. An example config file can be seen in table 3.3

3.3 Access pattern modules

This section outlines the implementation of an access pattern module. The model for simulating a generic I/O load presented in section 3.2 will be implemented with a number of specific functions. Some of them will be timed by the harness, while others will not.

Example:
PIOB_access_functions simple_strided_functions =
{
init_simple_strided_module,
setup_test_stripes,
create_stripe_data,
write_stripe_unit_work,
verify_stripe_was_written,
cleanup_stripe_test,
cleanup_simple_strided_module,
};

Table 3.4: Access pattern module registered information.

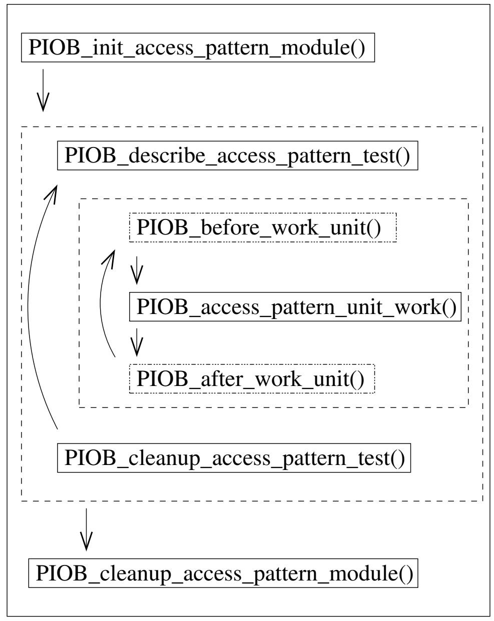
Example:
PIOB_access_table simple_strided_table =
{
&simple_strided_functions,
"Simple strided (write)",
};

Table 3.5: Access pattern module registered information.

3.3.1 Overview

The timing harness allows for the rapid substitution of a number of different access pattern modules. This will allow for modularity and enable the use of a standardized timing and reporting structure. Code duplication is also kept to a minimum inside the harness since repeated elements (such as timing infrastructure) can be included once by the main timing harness. This eliminates the need for an access pattern to handle communication for timing, for example. As a result, an access pattern module is free to perform file access and does not need to be complicated with timing code. Each access pattern module must pass certain pieces of information back to the timing harness. For example, it is necessary for the timing harness to know the name of the access pattern module, as well as a structure full of the function pointers to accomplish certain elements of access. See table 3.4 for the way to build a list of functions that comprise an access pattern module. Each access

Figure 3.4: Access pattern module flow



pattern module will specify at least five of the access pattern function pointers as shown in the table. This structure must be able to be accessed by the harness. In addition to the function pointers, the name of the access pattern module should be made available to the timing harness. See table 3.5 for an example of this. The structure of access pattern module functions, as well as the name of the access pattern module are the two key pieces of information that an access pattern must provide in order for the timing harness to know about it.

3.3.2 Methods

As shown in figure 3.4 here is an API to represent an access pattern module. Following the model that was proposed earlier in this chapter, we have generalized file I/O into several phases. The following methods map to portions of the phases defined above. We propose that this model will allow maximal standardization of timing and reporting while maintaining flexibility:

- **PIOB_init_access_pattern_module(...)**: This method initializes the access pattern module, can read in options from a configuration file, can set up any files for read tests, and must provide some basic information back to the test framework. Since this is the first method that the timing harness calls, it is responsible for setting up key bits of information that are required to start running access pattern tests from a module.

  An access pattern module can run multiple access pattern tests. An access pattern test is classified as the combination of run time parameters such as buffer size, offset into the file, and number of work units to be completed. This method is responsible for obtaining these options from some source. Whether the run time parameters are hard-coded, read from file, or interactively gathered by some other means, this method is responsible for supplying them to the timing framework.

  After this method has completed, the timing framework knows the buffer sizes, header offsets, total work units per test, and number of tests that will be run. This method is also responsible for providing MPI hints that will be used to enable optimizations from an MPI-IO level for each access pattern test. This method is not timed by the harness. Please see table A.1 which is located in appendix A for the parameter structure this method is passed.

- **PIOB_describe_access_pattern_test(...)**: There can be many access pattern tests run as part of an access pattern module, and this method is called by the timing framework to start a new access pattern test. There can be a one to many relationship between an access pattern module and an access pattern test. Therefore, this method may be called multiple times after an access pattern module has been initialized.

  Each access pattern test may consist of multiple "work units" which is where file access takes place. By the time this function is called, we have a valid handle to the file under test. This is useful should the access pattern need to setup a file view.

Similarly, the file has been opened with any hints that were supplied by the module initialization method.

In addition to file view management, this function is responsible for setting up any memory buffers and MPI datatypes that the access pattern may need (IE: to describe memory and file regions of access). Buffer size as well as whether contiguous or non-contiguous access is to be performed will influence this setup. This method is not timed. Additionally, this method is optional since not all access pattern modules need to perform anything on a per work unit basis. Please see table A.1 which is located in appendix A for the parameter structure this method is passed.

- **PIOB_before_work_unit(...)**: This method should accomplish anything that needs to be done before the work unit, such as seeking to the correct position, resetting a file view, or accessing other areas of the file to minimize the effects of memory paging or caching by the underlying filesystem. This function is not timed. Please see table A.1 which is located in appendix A for the parameter structure this method is passed.

- **PIOB_access_pattern_unit_work(...)**: The work unit function performs the actual filesystem access (a read or a write or sequence of reads or writes). This function is strictly timed according to the scheme laid out in section 3.2.1. Since this important element is timed, it is necessary for an access pattern test to only perform operations that count in this method. Any blocking calls other than flushing data to disk should be called elsewhere.

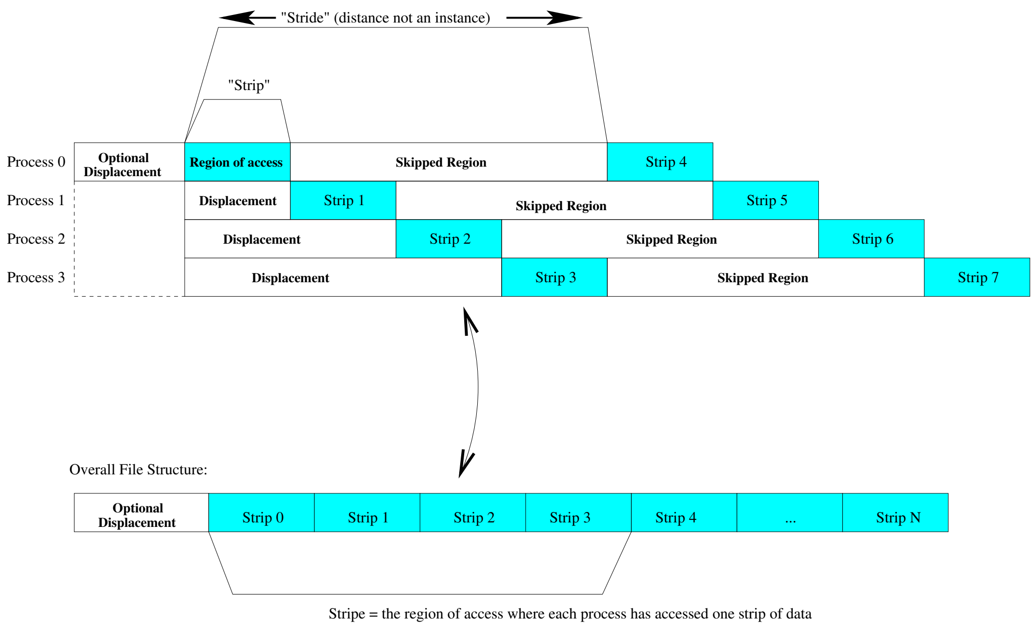
  This function is responsible for respecting write flushing. The timing framework will indicate whether the user requested data to be flushed. Since an access pattern module is the only thing that knows whether its going to read or write, handling flushes is not a reasonable thing for the timing framework to handle. While it could be a useful simplification, it is best left for the access pattern module to respect or ignore those configuration directives. Please see table A.1 which is located in appendix A for the parameter structure this method is passed.

- **PIOB_after_work_unit(...)**: This method is called after the timed work unit method has completed. This execution time is not counted in the total execution time for the corresponding work unit. This method is useful for verification, or for cache manipulation operations that may be costly, yet should remain untimed. This method is optional since not every access pattern module will need to perform anything on a per work unit basis. Please see table A.1 which is located in appendix A for the parameter structure this method is passed.

- **PIOB_cleanup_access_pattern_test(...)**: This method is called after all the work units for a given access pattern test have completed. It is responsible for any cleanup required for the next access pattern test to execute. If a sequence of writes occurred during this access pattern test, it may be necessary to delete the file under test (since the next access pattern test will need to write to the same filename). The file has been closed at this point, and any buffers used during this access pattern test should be deallocated. Any datatypes that were used should also be released via the MPI_Type_free() call. This method is not timed. Please see table A.1 which is located in appendix A for the parameter structure this method is passed.

- **PIOB_cleanup_access_pattern_module(...)**: This method is called by the timing framework after the entire group of access pattern tests has run to completion. This method should free any remaining memory that was used to hold configuration data. Deleting the file under test may also be done in this method. Depending on the access pattern module, the file under test should either be deleted by this method, or the cleanup method described in the previous bullet. This method is not timed. Please see table A.1 which is located in appendix A for the parameter structure this method is passed.

Figure 3.5: Simple strided access pattern



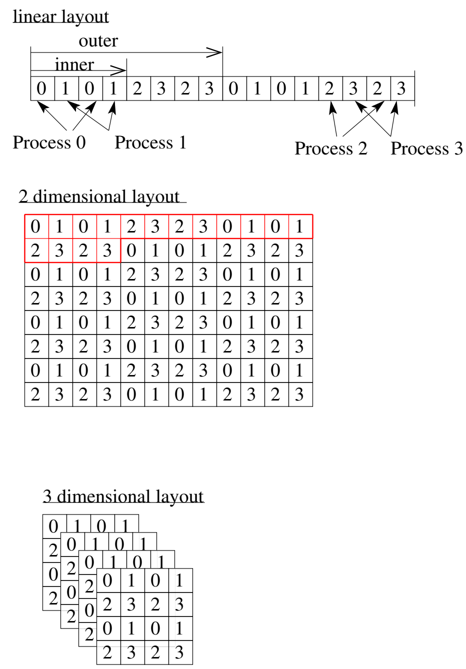
3.4 Access pattern scope

In this document, when the term "Access pattern" is used, it refers to the specific mapping that exists between an MPI process and some set of bytes within a file. The relationship between the file and the process will be described in terms of spatial organization as well as temporal ordering. This twofold relationship gives us a broad range to cover. Not only must the access pattern describe the file in terms of which bytes were accessed, but also when they were accessed. This broad definition will help to differentiate between behaviors such as "read once" and ”re-read”. There are patterns which are similar but distinct in their temporal ordering.

3.4.1 Spatial access patterns

Here are a number of common access patterns that have been collected for study. Each of the following access patterns has been implemented using the model for I/O presented in section 3.3.2:

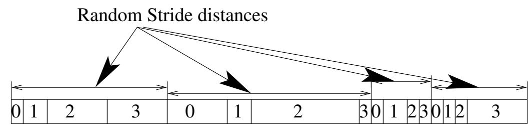
Figure 3.6: Nested strided access pattern



- **Simple strided**: The simple strided access pattern divides sections of the file into logical stripes where each process will access bytes at some displacement relative to a stripe. The displacement of a process into a stripe typically is related to its rank in the set of processes. All the regions of access within a stripe are called strips, and the sum of the length of all the strips is referred to as the "stride distance." Each stripe within a file occurs regularly at some multiplier of the stride distance. Please refer to Figure 3.5 for a graphical representation of the simple strided access pattern. Possible applications of simple strided are unlimited as it has been shown to be the most widely used access pattern accounting for roughly 63% of file access in several scientific applications [24, 4].

- **Nested strided**: Nested strided is a complex access pattern that combines multiple simple strided accesses into each striping element. Namely, each strip of access within the nested strided access pattern is one or more stripes of access of a simple strided access. There is an internal stride distance and an external stride distance for

Figure 3.7: Random strided access pattern

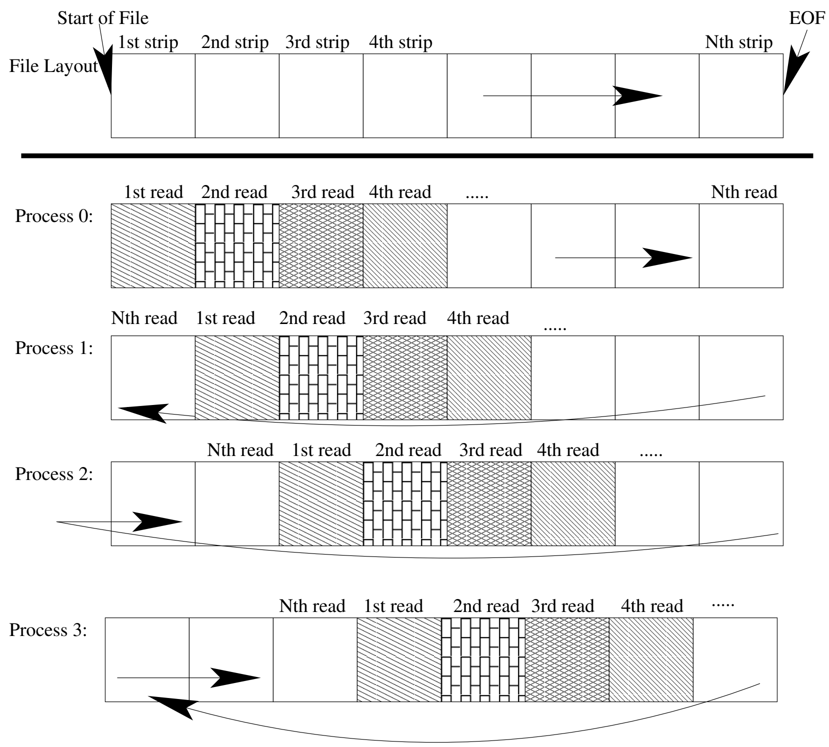


each region of access within a file. This type of access can be represented in two dimensional form as a series of tiles. Please see figure 3.6 for a graphical representation of the nested strided access pattern. Possible applications of the nested strided access pattern include accessing elements of a three dimensional cube inside of a file. The X and Y dimensional offsets may be placed such that a simple strided pattern would be unable to access any of the Z dimensions elements of a cube. Also, according to workload studies, nested strided access accounted for roughly 33% of requests for a variety of scientific workloads [24].

- **Random strided**: Random strided is categorized by having each process access some non-zero number of bytes in an ordered round robin fashion. Since each process is responsible for accessing an non-zero number of bytes, the stripe size for each cycle varies. This results in a random stride distance for each iteration of accesses. See figure 3.7 for a graphical representation of the random strided access pattern. Possible applications that utilize the random strided access pattern would be media encoding applications where each frame size is variable. If each process generates a variable sized frame and then writes it out to file after the preceding frame, that would simulate the type of access shown in this pattern.

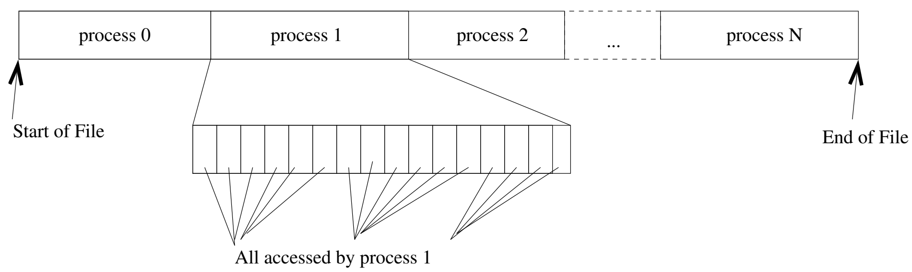
- **Sequential access**: Each process opens the file and issues requests that access a fixed size region with linearly increasing offsets for each successive request in a sequential access pattern. The requests start at the beginning of the file, and continue until each process reaches the end of file. Optionally, a user may specify an offset that will cause each process to essentially time-shift its access of the same regions of the file. This

Figure 3.8: Sequential read



allows a process to interleave its access such that it is not accessing the same strip of data of a file as any other process. This approach will allow for maximum throughput for file systems that allow regions of a file to be partitioned on separate I/O servers. This access pattern can be decomposed to a simple strided access pattern where each process' strip size is the same, and each process has no displacement relative to the beginning of a stripe. Please refer to figure 3.8 for a graphical representation of a sequential read. Sequential reading makes sense (as all processes can simply read the same file), but sequential writing is a different issue altogether. Having multiple writers potentially accessing the same regions of a file does not make sense due to the fact that whatever data written would be there as the result of a race condition, namely whatever writer was the last to write to an area is the one whose data remains there. No real applications intentionally invoke such race conditions to the same

Figure 3.9: Segmented access pattern

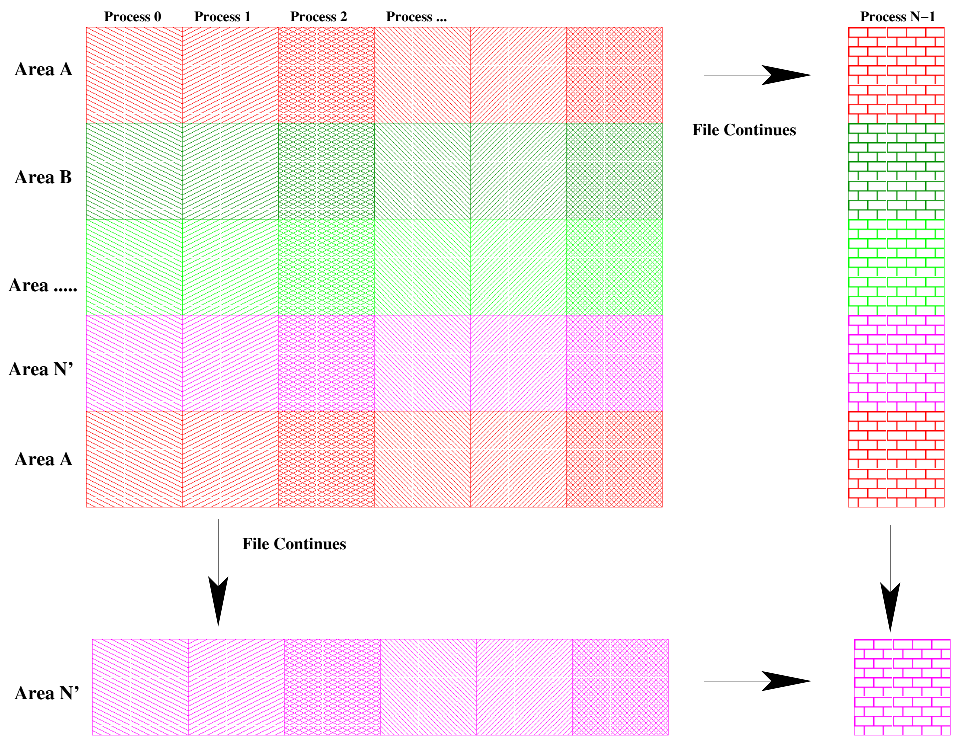


region, so it is necessary to simulate sequential writing differently. That is why each process writes to a different file with a sequential access pattern.

- **Segmented access**: The segmented access pattern divides the entire file into logical segment ranges for as many processes as there are performing concurrent access. These divisions are based upon logical byte ranges within the file. For example, if a file is 4096 bytes long and there are four processes, process 0 could be assigned bytes 0-1023, process 1 could be assigned bytes 1024-2047, process 2 could be assigned bytes 2048-3071, and process 3 could be assigned bytes 3072-4096. In this example, each process' region of access is determined by its rank. Ordering based upon rank is one example of segmented access, but a segmented access pattern may dole out regions of access by any means it chooses. Please refer to figure 3.9 for a graphical representation of the segmented access pattern.

- **Tiled**: Tiled access is categorized by each process accessing an area of the file which is viewed as a "virtual tile." Depending on the layout of the tiles, this pattern can either be decomposed to segmented access or simple strided access. See figure 3.10 for a graphical depiction of how processes interact with files in this access pattern. One example of this pattern in use is displaying a large virtual screen on multiple video devices. Since each tile can be mapped to an area of a larger unified region, this sort of access pattern is seen when large visualization environments need to record

Figure 3.10: Tiled I/O



or play back data. This is the main access pattern that the mpi-tile-io benchmark exercises [20].

- **Flash I/O Kernel**: The center for Astrophysical Thermonuclear Flashes (Flash center) at the University of Chicago studies many of the effects of thermonuclear explosions on the surfaces of many types of stars. In particular, simulations related to X-ray bursts of Type 1 supernovae provide a demanding workload for I/O resources [35]. The spatial access pattern that the Flash code utilizes is non-contiguous in memory and non-contiguous in file and has been the topic of I/O optimization studies [27]. Essentially, the access pattern is sequence of "memory blocks" where each memory block contains a three dimensional cube of data. Inside this three dimensional cube, there are areas that need to be accessed in file, and those that are skipped over. The skipped areas are referred to as "guard cells." While the scope of the physics simulation that Flash aims to explore is beyond the scope of this paper,

Figure 3.11: Flash I/O Memory Layout

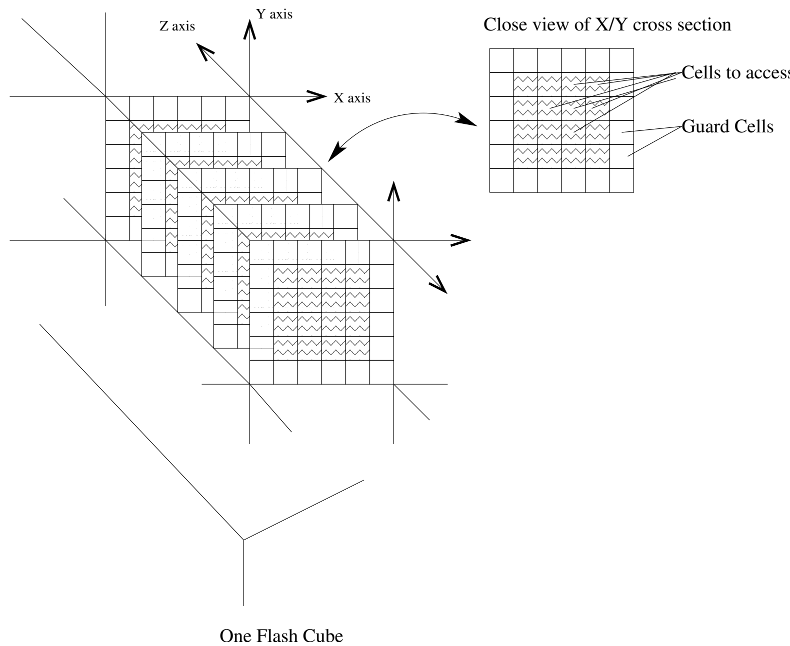
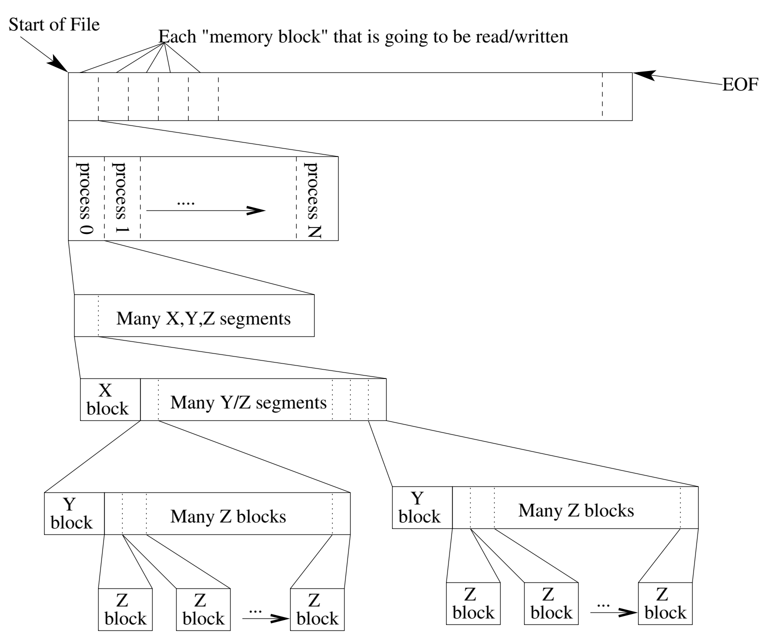


Figure 3.12: Flash I/O Disk Layout



its I/O counterpart is definitely something that is of interest. Please refer to figure 3.11 for a graphical description of how data is laid out in memory and figure 3.12 for a graphical description of how data is laid out in file.

- **Unstructured Mesh**: Applications that simulate terrain mappings, or wire-frame engineering models typically abuse parallel filesystems when they need to store or load their large datasets. The data that they need to store in a file is a collection of points that comprises a large number of polygons. Since the topology of terrain, or contour maps may vary immensely, these structures are referred to as "unstructured meshes." This data can be thought of as a point in three dimensional space with X, Y, and Z coordinates. Commonly these points will also have some other value, or structure of values for each point in space (a moment of inertia, a gravitational force, or, other forces about an axis, etc) [30]. These types of three dimensional spatial mappings are also seen in finite element approximations and computational fluid dynamics problems. Since these problems are partitioned on nodes throughout a cluster of machines, each node is working on its own subset of the large spatial simulation. This results in each process having to access a relatively small area for each point in the structural analysis problem (IE: 3 floating point numbers for X/Y/Z coordinates, and a structure describing forces on that point in space). These types of simulations typically can be decomposed into one of the access patterns described previously in this chapter. Namely, either a simple strided, or segmented access pattern will be used by a parallel application when it needs to store or load unstructured mesh data points from file.

3.4.2 Temporal access patterns

In addition to spatial access patterns, the mapping between an MPI process and a region of file may vary with regard to time. This will cause varying performance depending on several timing issues. The following are common ways to vary access with respect to time, and each may be combined with the spatial access patterns listed in section 3.4:

- **Read once**: This form of access is classified by retrieving a number of bytes from some location within a file to a location in memory. The buffers that the system fills with file data may be non-contiguous. Similarly, the file data may also be drawn from non-contiguous areas and copied to contiguous memory buffers. This access may be affected by caching, or prefetching, but any action as such would rely on some predictive mechanism at the filesystem level. For the purposes of this benchmark, all reads are assumed to be blocking. The contents of the file are assumed to have been moved to the appropriate memory buffers in their entirety by the time the read call returns.

- **Write once**: This form of access is classified by the storage of bytes from some location in memory to a location within a file. Typically these locations in the file are specified by an offset, and a buffer length. Since the data is in memory, and must be copied to the location within a file, a filesystem has the option of flushing the data to file immediately, or holding onto the data in an internal buffer until it is convenient to write to file. There are several factors that affect this decision such as block granularity and optimal network packet sizes. A filesystem that stores data on a remote server more than likely will incur both of these expenses if a large number of requests with small pieces of data are received. Forced flushing may lead to slower performance than is seen without explicit file synchronization. However, filesystems that offer non-blocking writes as the default may need to have data explicitly flushed in order to ensure that it has reached the file.

- **Read-modify-write**: This form of access is classified by the retrieval of a number of bytes from a specific location within a file, each of those bytes will be changed, and then the new values will be transferred back to the location in the file from which they were originally read. This is the most intensive temporal access pattern due to the fact that the filesystem has to retrieve an up-to-date copy of the file, and then push those bytes back out to file. This is important since many out-of-core simulations which use filesystems as their main computational space utilize this access pattern extensively as they manipulate data. Read-modify-write can take on many forms, such as reading the entire file in its entirety before writing, or reading each buffer and instantly writing.

- **Re-read**: This form of access is classified by a process retrieving a specific number of bytes from a specific location within a file to a memory location, and then some time later, the same location from the file is transferred to memory again. The second read is the only operation timed in this mode of access. Unless the region of file has been changed by a transient write that occurred after the first read, the data in that location of the file should be the same as it was at the time of the first read. This would show good performance for a filesystem that is able to intelligently decide whether to supply data from a cache, or to transfer the entire buffer from file across the network.

- **Re-write**: This form of access is classified by two consecutive storages of some set of bytes in memory to a specific location within a file. The second set of bytes transferred to file does not need to be independent of the first set of bytes. This access pattern will showcase any filesystems that may offer write-behind caching. Since writes are often costly, some filesystems offer a caching mechanism that waits for a short period of time to move the buffer from memory to file. On the other hand, it is possible that the second write would be slower if the data is not aligned on block boundaries.

3.5 Summary

Using lessons learned from previous benchmarking efforts, the preceding chapter has presented the design of a performance evaluation standard that should allow for a large amount of performance data to be retrieved from a parallel filesystem. Through the use of the generic model of I/O presented in section 3.2, a number of access pattern modules have been implemented. Because the timing framework calls the work unit method for each access pattern test, each access pattern module uses the same timing code. This provides the critical simplification to allow consistent, comparable results. This technique eliminates disparities due to differences in timing code because the framework uses the same timing code to monitor each access pattern module. After establishing a timing model, it was necessary to figure out what should be timed. Section 3.3.2 presents the seven functions that comprise an access pattern module. Having established a timing mechanism and a modular framework with which to implement access patterns, section 3.4 presented a large number of spatial access patterns. In addition to spatial access patterns, temporal access patterns were also discussed in section 3.4.2. Furthermore, these access patterns define a relationship between a process and a file, and this definition can be described in terms of time ordering. In the following chapter, the implementation of these modules, and the timing harness will be examined in depth.

Chapter 4

Results

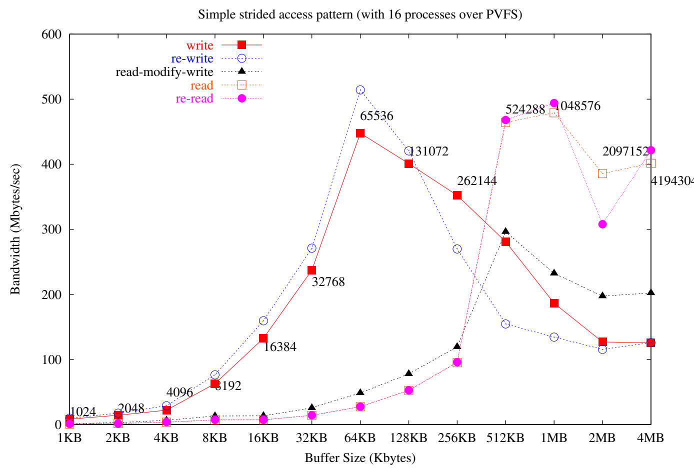
4.1 Overview

This section will present actual numbers provided by this benchmark, as well as begin to analyze the effectiveness of this implementation and its adherence to the qualities set forth in section 1.4. The pio-bench benchmark will be examined with regards to how well it applies to the principles of software engineering. There will be some focus on the actual results that this benchmark is capable of gathering, but only as a means to emphasize comparability, and comprehensiveness. The following sections will describe the test environment that this benchmark ran on, what trends were returned for each access pattern module, as well as discussing any particular implementation issues that needed to be solved while these benchmarks were in the process of being developed.

4.2 Access pattern test results with PVFS

The following sections will report the results that were obtained from each access pattern while running on PVFS.

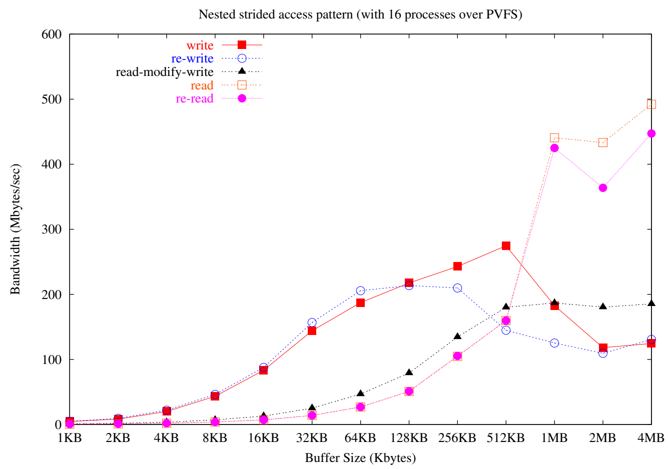
Figure 4.1: Simple strided performance



4.2.1 Test environment

After implementing the code, it was run with each access pattern module enabled on a 16 node development cluster. Each node contains dual Intel Pentium III 1 GHz processors with 1 gigabyte of RAM and 2 Maxtor 30 gigabyte hard disks (model number 5T030H3). An Intel Etherexpress pro 10/100 Fast Ethernet card and a high speed Myrinet network card comprise the networking connectivity of each node. Red Hat Linux 8.0 with the Linux 2.4.20 kernel was running at the time of this testing. Version 1.5.7 of PVFS was running with four mounted filesystems total. There were two filesystems running with 16 IODs, one ran over Myrinet, the other over Fast Ethernet. The other two filesystems are running with four IODs on four nodes. The head node ran one MGR process to manage all four filesystems. The PVFS striping size for each filesystem has been set at 64 KB (65535 bytes). MPICH 1.2.5.1a was used for all of the tests. For the graphs shown below, each access pattern test was run 50 times, and the results from each run were averaged together.

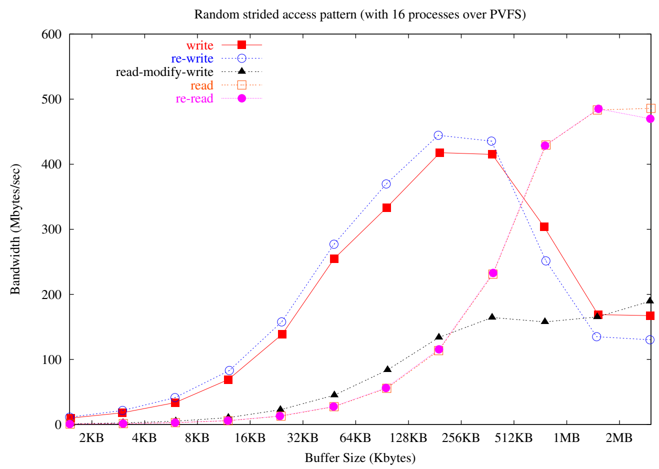
Figure 4.2: Nested strided performance



4.2.2 Simple strided

Figure 4.1 shows the performance that PVFS provides for read, write, read-modify-write, re-read, and re-write with a simple strided access pattern. As the graph indicates, maximal read performance was obtained with a buffer size of 1 megabyte. Maximal write performance was achieved at the PVFS striping size of 64KB. Re-read performance was consistent with read performance, except at the 2 MB buffer size, the second read is noticeably slower. In the read-modify-write case, it seems that the most costly operation (the read call, or the write call) is not the upper bounds upon the transfer speed for a given buffer size. This can be explained by the fact that the benchmark is reading and writing, which moves twice the amount of data. When averaged, the resulting value will be somewhere between the upper and lower trend lines (depending on the relative difference in magnitudes). As far as the general trends go, this access pattern is the best overall performer on PVFS, which is no surprise given the nature of how PVFS stripes data across multiple IODs.

Figure 4.3: Random strided performance



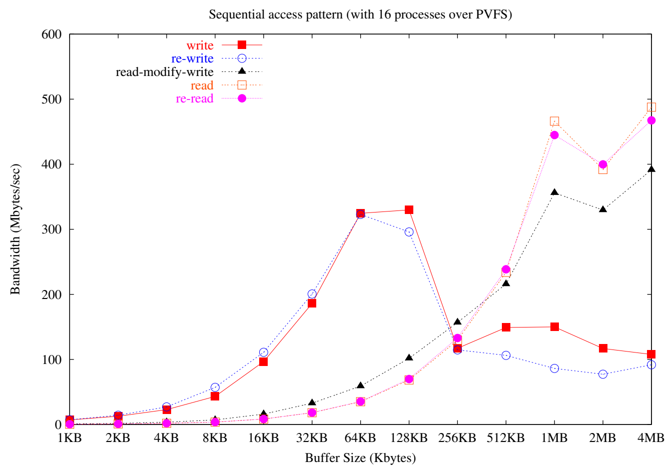
4.2.3 Nested strided

Figure 4.2 shows the performance that PVFS provides for read, write, read-modify-write, re-read, and re-write with a nested strided access pattern. As seen in the graph, the magnitude of read and write throughput is not as fast as with a simple strided access pattern. While this access pattern does not yield performance as good as simple strided, the trends seem to indicate that same things that affect bandwidth rates with a simple strided access pattern, also affect bandwidth with a nested strided access pattern.

4.2.4 Random strided

Figure 4.3 shows the performance that PVFS provides for read, write, read-modify-write, re-read, and re-write with a random strided access pattern. For this access pattern test, random strip sizes were generated for each process during each run. The random strip sizes were bounded between regular intervals for each run (for example: test one was between 1024-2048 bytes, test two was between 2048-4096 bytes, test three was between 4096-8192 bytes, etc). The points that are plotted on each trend line are at the average buffer size for each test. While there are varying buffer sizes between an upper and lower bounds, this

Figure 4.4: Sequential performance

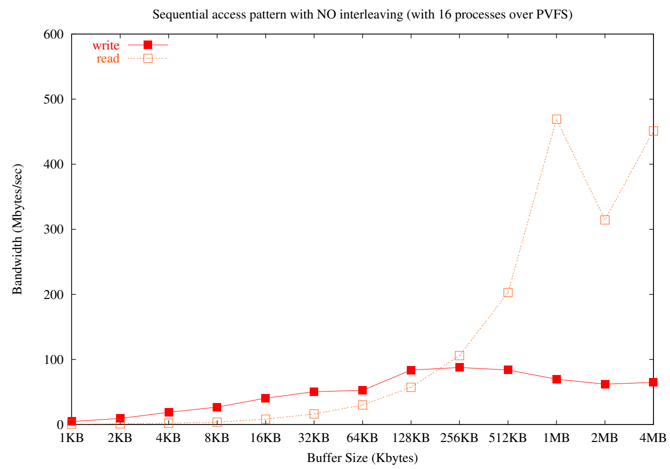


average strip size is the average of each strip that every process accessed. As is shown in the graph, random stride distances appear to produce the same trends as a simple strided access pattern even though the performance numbers are obviously not at the same magnitude.

4.2.5 Sequential

Figure 4.4 shows the performance that PVFS provides for read, write, read-modify-write, re-read, and re-write with a sequential access pattern. While sequential access seems straightforward, there are ways to time shift sequential access on certain nodes to alter performance. Namely, if a filesystem has portions of the file partitioned on independent serving devices (the way PVFS has multiple IODs managing portions of a single file) it greatly improves performance to stagger access to the same portion of a file. This can be seen with the contrasting performance in figure 4.5. Time shifting access to different regions of the file essentially makes sequential reading as speedy as simple strided access on PVFS. When each process is accessing the same section of file at once, each node must contact the same IOD to retrieve that area of the file. This will result in a bottleneck at each IOD where aggregate throughput will be limited by the maximum bandwidth that a single node can put out. As shown in the graph, when the buffer size used to access the file grows

Figure 4.5: Sequential performance (no interleaving)



larger than the PVFS stripe size (64 KB), each process will be accessing regions of the file that reside on multiple IODs. This results in a slight increase in throughput for accesses that are larger than the PVFS striping size.

4.2.6 Segmented

Figure 4.6 shows the performance that PVFS provides for read, write, read-modify-write, re-read, and re-write with a segmented access pattern. Segmented access involves logically dividing up regions of a file such that each process has a certain logical range of bytes. For filesystems that stripe files across multiple nodes, this means that in order for a process running on a node to access each byte in its region, it will have to access several areas that are stored on remote servers. The resulting performance of a number of non-local accesses will be upper bounded by network performance. This behavior can be seen in the performance graph as writes do not improve by any large amount as buffer sizes increase.

Figure 4.6: Segmented performance

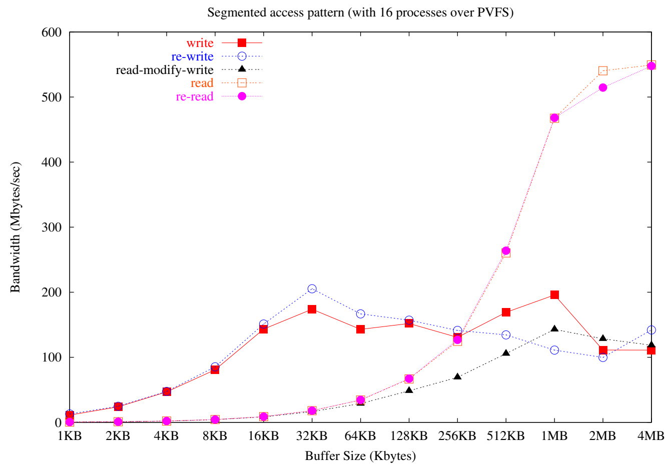


Figure 4.7: Tiled performance

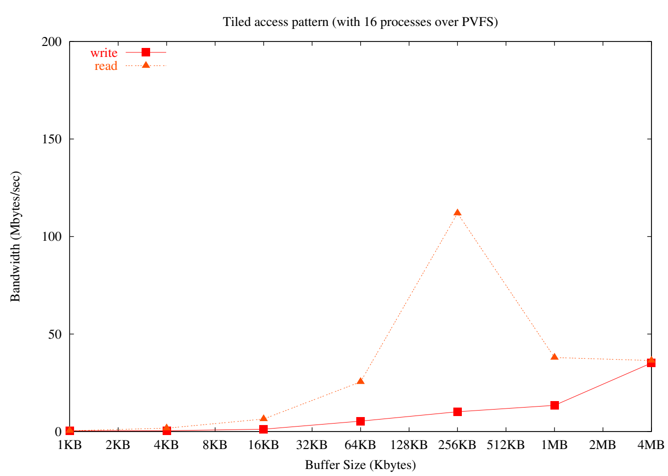
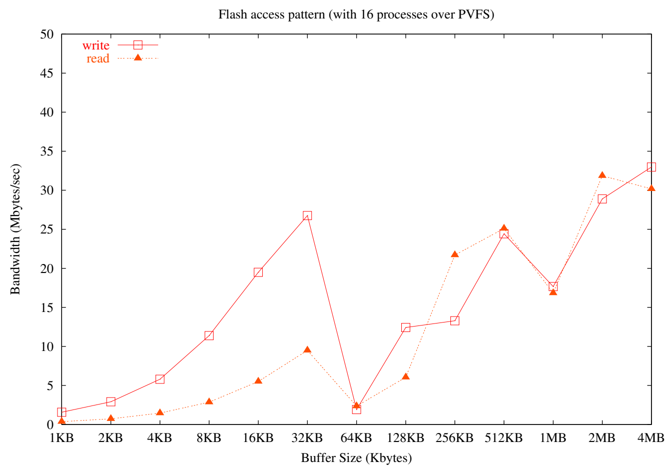


Figure 4.8: Flash performance



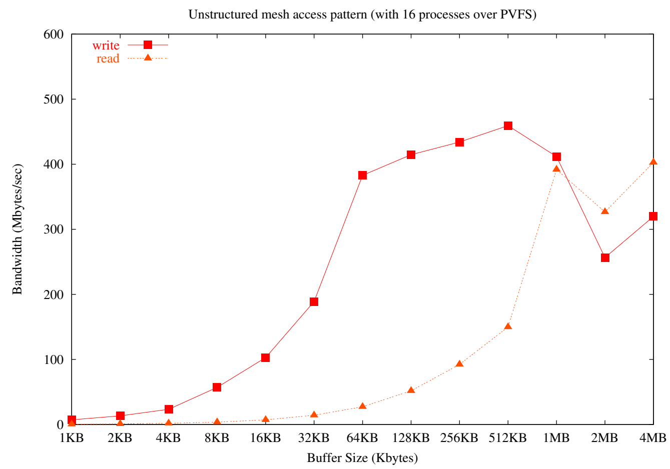
4.2.7 Tiled I/O

Figure 4.7 shows the performance that PVFS provides for read and write with a tiled access pattern. While tiled access is similar to simple strided in how it accesses regions of files, it has its own distinctive characteristics while performing tiled access. Tiled performance on PVFS is not close to the magnitude of performance that simple strided access achieves. It seems that this is due in part to the mapping of bytes in a tile to its physical location within a file. In tiled access, each process is assigned a certain tile, or number of tiles to access. The mapping of logical bytes within a tile to physical bytes within a file may not best fit the striping unit that an underlying filesystem is best suited to handle. Instead of each tile being mapped to the appropriate stripe size, the tiles are laid out logically according to their order within the file. When these regions are mapped to an area of a file, the striping unit that is chosen is clearly not the factor being manipulated directly by the testing harness. While performance is somewhat better than if it had been run on a single node for higher tile sizes, these trends may not be representative of the potential of tiled access.

4.2.8 Flash I/O

Figure 4.8 shows the performance that PVFS provides for read and write with the Flash I/O access pattern. Flash I/O heavily exercises the ability of a filesystem to provide non-contiguous file access. As seen in the results, increasing buffer size results in a step-like pattern showing an increase in transfer rates. This is due to how the flash blocks are represented in memory. Please refer to figure 3.12 for the layout of data on disk and figure 3.11 for the layout in memory. Both the file and memory datatypes are non-contiguous for this access pattern. As the buffer sizes are increased, the contiguous area that each process accesses varies irregularly. What we can see from the flash performance is that as buffer sizes are increased, the throughput drops once it reaches a certain point. Transfer rates monotonically increase for a few data points, but then decrease again. The number of non-contiguous areas that must be accessed increases at the point where the transfer rate decreases on the graphs. The flash blocks contain a number of cells in the X, Y, and Z directions. In order to increase the amount of data written, one can either increase the number of cells per memory block, increase the number of memory blocks, decrease the number of guard cells, or increase the number of elements per cell. If more memory blocks are added, this results in more non-contiguous areas that a filesystem has to access. On the other hand, increasing the number of elements per cell, increasing the number of cells per memory block, or decreasing the number of guard cells will increase the size of any contiguous region that must be accessed. What we see is that increasing the number of non-contiguous regions will negatively impact performance, while increasing the size of a contiguous region will increase performance. It would be better to refer to the amount of data written as "transfer amount" or "transfer size" or something other than buffer size. The name buffer size leads one to believe that there is only one buffer, that it is contiguous, and of a certain size. Since this is not the case, this is a candidate for a name clarification.

Figure 4.9: Unstructured mesh performance



4.2.9 Unstructured mesh

Figure 4.9 shows the performance that PVFS provides for read and write for an unstructured mesh access pattern. Buffer sizes in this graph correspond to the number of verticies, as well as the number of values per vertex. Since the values associated with each vertex are packed into a contiguous buffer varying the number of verticies or values per vertex simply increases the amount of contiguous data that should be accessed by a process. As shown by the graph, as buffer size increases, performance also increases. The performance for this access pattern is similar to the performance for simple strided even though the actual pattern more closely resembles the segmented access pattern.

4.3 Implementation issues

Throughout the implementation of this system, there were multiple goals that needed to be achieved in order to make this benchmark usable. Since the access pattern modules needed to be able to be added in a manner that would allow many patterns, the pattern API as described in section 3.3.2 was needed. Every access pattern was written in C and uses MPI-IO to access files, this ensures portability from machine to machine, and portability across heterogeneous environments. As shown from the large amount of graphs from the results, this tool can produce a plethora of results from the number of access pattern modules that exist. Configuration is done on a per access pattern basis via text files. This is advantageous for testing multiple configurations rapidly. Since it is possible to save the module configuration and simply comment out the lines that do not apply, there is relatively little effort involved in choosing an arbitrary access pattern module or choosing parameters for that module.

4.4 Summary of results

In the preceding sections, the results of the pio-bench evaluation tool for parallel filesystems have been presented. Each access pattern module that was implemented has been graphed in section 4. These graphs make it possible to observe trends among several access patterns, as well as understanding why accesses perform as such. In addition to providing comprehensive data about a number of widely used access patterns, implementation issues were presented that illustrate some of the challenges that were overcome while implementing the timing framework and each access pattern module.

Chapter 5

Conclusion

In the previous chapters, we have presented and analyzed a tool for evaluating parallel I/O. The following sections will wrap up some key points and examine how this tool measures up to the software engineering goals introduced in section 1.4.

5.1 Comparability

As shown in the results chapter, this benchmark is useful for comparing many types of parallel access. However, this comparability is limited by the access pattern model that has been adopted. This model works well for contiguous accesses, but extreme care must be used when comparing file accesses that are non-contiguous. The timing framework only understands the idea of a "buffer size" and as buffer sizes are increased, certain patterns follow certain trends. With non-contiguous file access, it is possible to manipulate certain runtime parameters to map to the same buffer size even though the amount of data written has nothing to do with the access pattern distribution. For example, the Flash I/O module accesses segments of file that correspond to three-dimensional cubes with a border ("guard cells"). The runtime parameters comprise the number of blocks in each direction, and the number of elements that should be accessed per block. If the number of elements accessed per block is increased, a large contiguous area is accessed. If the number of X, Y, or Z blocks is increased, this will result in a number of non-contiguous areas that should be accessed. Through careful manipulation of these parameters, it is possible to construct differences in performance with the same amount of data being transferred. Namely, accessing a large number of small, non-contiguous regions performs worse than accessing a small number of large, non-contiguous regions. Both can be achieved with the same access pattern module, and both map to the same buffer size. This detracts somewhat from the comparability because it is necessary to explicitly state the run-time parameters for this access pattern module. This caveat is primarily a terminology issue, and it requires that "buffer size" be understood as the amount of data transferred by a work unit, rather than the size of the segments of data that are stored within a file. This terminology choice limits the implementation of access patterns that rely heavily on non-contiguous access patterns. They can still be accomplished with this model, but it is necessary that the access pattern module fully disclose what each runtime parameter controls.

5.2 Portability

As stated in the design chapter, this benchmark uses MPI for all communication, and MPI-IO for all access pattern I/O calls. Combined with C that is GNU compliant, this benchmark should compile on any platform that offers MPI support. The use of MPI as a communication library ensures that all messages sent to each process will be converted to a known format before being sent over the wire. This is key to operating in a heterogeneous environment, and it is handled by the mpi implementation (MPICH in this case).

5.3 Comprehensiveness

As shown in the access pattern results section, a large amount of data is returned about the performance of a given filesystem. Some of the most commonly used access patterns have been implemented, and there are many parameters that can be adjusted for each of these patterns. While it is obvious that these access patterns do not constitute the entire set of possible access patterns, they are representative of the file access that is performed by a number of parallel applications. Since benchmarking is aimed at trying to simulate workloads without the time, energy or effort of running an entire simulation, it is sufficient to model parallel file access as the set of implemented access patterns do. However, should the representative file access patterns change, this tool is flexible enough to change to meet those needs as well. Through a well defined, extensible model, we have shown that it is possible to simulate a number of access patterns while maintaining consistent timing and reporting standards. It is possible to add access pattern modules to increase coverage should the need arise.

5.4 Objectivity

The access patterns gathered by this benchmarking effort have been obtained by examining workload characteristic papers, and submissions from applications programmers. While this benchmarking effort clearly gives more coverage to the obvious and widely used access patterns, it is relatively easy to add new access patterns. Also, at this time, the framework does no weighting of any of the access pattern workloads.

5.5 Future work

While this tool provides a large amount of data about the performance of a given parallel filesystem under certain workloads, it does little to eliminate the subjective nature of evaluation. At some point the results must be interpreted, and different weights should be applied to different workloads that are consistent with the individual needs of a given application.

5.5.1 Expansion of access patterns

Obviously this access pattern list is not all inclusive. The access patterns that we have included here are general access pattern cases which were gathered from the behavior of parallel applications that are being used by the scientific community. As well as the collection of general use patterns, there are also 3 specialized patterns that showcase I/O performance based upon specific application requirements. The tiled I/O, Flash, and unstructured mesh access patterns are examples of how dissimilar access patterns have been integrated into this common timing and reporting framework. They were submitted as case studies when the application developers were having problems with poor performance. This type of feedback is the only way to accurately gauge the I/O demands of a real application. It has been shown that this standard timing and reporting framework is capable if integrating multiple dissimilar access patterns, and the hope is that application developers will share their I/O requirements at a high level so that these needs can be studied and optimized. Upon receiving a high level understanding of an application's I/O needs, it will be relatively simple to assimilate more access patterns into this common framework.

5.5.2 Framework for interpreting results

As shown by the results section, PIO-bench is capable of gathering a large amount of data about the performance of a parallel filesystem; however, there is still a subjective element involved in analyzing performance. While this benchmark can presently gather a large amount of data from an underlying filesystem, interpreting the results can be difficult to understand at first glance.

A system to weight access patterns that are more important to specific applications would be advantageous in that it would make results easier to view. Since different parallel applications use a variety of patterns to access data, choosing the right filesystem/filesystem configuration can be difficult if one is not aware of how the system will be used. Namely, such a weighting system would stand to highlight performance of access patterns that are commonly used by specific applications. This feature could be more of a presentation style issue as opposed to a feature that weighs results and averages data from multiple tests. PIO-bench returns a deluge of quantitative information about the performance of a parallel filesystem, but sifting through it can be troublesome.

When considering a weighting algorithm, it is necessary to consider several things. What kind of access pattern do applications that run on the system under test typically use? What kinds of trends can be observed in each of the patterns? Is it possible to reduce data from each access pattern test and each access pattern module to a single numerical ranking? How diverse is the set of access patterns that this benchmark comprises? Does the diversity affect the ability to make generalizations about the performance?

A performance analysis component should allow for the explanation of trends that are observed by the performance of a certain filesystem on a certain access pattern. For example, as demonstrated in section 4.2.2, the simple strided access pattern achieves the best throughput with buffer sizes of 64 KB. This can be explained by the fact that PVFS was compiled with a striping size of 64 KB for these test runs. The striping size controls how large of a strip of data is stored on each server. If the striping size were changed to 512 KB, write performance should change to reflect this. Such characteristics (striping sizes of de-clustered regions of a file) will be evident from a visual inspection of the performance data trend line, but coming up with a set of obvious characteristics is left open for future work to describe.

In addition to a weighting system, there should also be a visualization system to automatically make graphs of access pattern performance. This could take the form of anything from a simple shell script to interface with gnuplot, to something that uses openGL and displays the access pattern being executed as the results are returned. The timing harness could easily be modified to store the results from each access pattern module to a text file, and then have a shell script to parse the results, and plot different series via gnuplot. This type of visualization will depend on any type of weighting and analysis interpretation.

For example, if performance data is to be compared from multiple system across multiple buffer/window sizes, it may not be feasible to plot data on a two dimensional series if it is cluttered.

Appendices

Appendix A

Access pattern module method parameters

Module initialization parameters	Description
struct pio_ap_init_params_s {	
int *mode;	Mode to open the file with (read/write/truncate).
int *communicator_type;	How to open the file (collective/shared).
int *num_reps;	How many tests are in this module?
char* file_name;	This is the filename provided in piobench.conf.
int *use_separate_files;	Flag to make the tests in this module use separate files.
int *num_procs_participating;	How many processes are doing actual I/O.
long long *num_work_units;	Array containing the number of work units for each test.
long long *displacements;	Array containing the amount of header bytes for each pattern test.
long long *buffer_sizes;	Array containing the buffer sizes for each access pattern test.
MPI_Info *nfo_array;	Array containing an ordered list of hints for each pattern test
};	

Table A.1: Access pattern module initialization parameters.

Test initialization parameters	Description
struct pio_datatype_init_params_s {	
long long header_bytes;	Amount of bytes that should be skipped before the first I/O access.
long long buffer_size;	Amount of bytes that this process should access.
long long num_reps;	Number of work units in this pattern test.
MPI_File *fh;	File handle to the unit under test.
};	

Table A.2: Access pattern test initialization parameters.

Work unit parameters	Description
struct pio_ap_work_unit_params_s {	
MPI_File fh;	File handle to the unit under test.
long long header_bytes;	Number of bytes to skip before this access.
long long buffer_size;	Number of bytes this access should touch.
long long rep_inst;	Number of the current work unit.
long long num_work_units;	Total number of work units.
int rank;	This process' rank.
long long size;	Number of processes.
int should_flush;	Flag indicating whether the user has requested flushing.
int use_collective;	Flag indicating whether the user would like to use collective I/O.
};	

Table A.3: Access pattern work unit parameters.

Cleanup parameters	Description
struct pio_cleanup_params_s {	
char* file_name;	Name of the file under test.
};	

Table A.4: Access pattern cleanup parameters.

Bibliography

[1] Donald J. Becker, Thomas Sterling, John E. Dorband Daniel Savarese, Udaya A. Ranawak, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Processing*, 1995.

[2] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[3] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, 34(2):222–248, January 1995.

[4] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[5] Thomas Sterling et al. *Beowulf Cluster Computing with Linux*. MIT Press, Cambridge, MA, 2002.

[6] H. Goldstine and J. von Neumann. the principle of large-scale computing, 1963.

[7] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[8] SIO Working group et al. Preliminary survey of i/o intensive applications.

[9] R. L. Haskin. Tiger Shark — A scalable file system for multimedia. *IBM Journal of Research and Development*, 42(2):185–197, March 1998.

[10] John E. Howland. Software freedom, open software and the undergraduate computer science curriculum.

[11] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)–part 1: System application program interface (API) [C language], 1996 edition.

[12] A. E. Koniges, R. Rabenseifner, and K. Solchenbach. Benchmark design for characterization of balanced High-Performance architectures. pages 196–197.

[13] Linpack. `http://www.netlib.org/benchmark/hpl/`.

[14] Lustre. http://www.lustre.org.

[15] John M. May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann, San Diego, CA, 2001.

[16] Larry W. McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[17] Message Passing Interface Forum. MPI documents. http://www.mpi-forum.org/docs/docs.html.

[18] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. http://www.mpi-forum.org/docs/docs.html.

[19] MPI-2: Extensions to the message-passing interface. The MPI Forum, July 1997.

[20] mpi-tile-io. http://www-unix.mcs.anl.gov/ rross/pio-benchmark/.

[21] Nas parallel benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[22] Nils Nieuwejaar and David Kotz. A multiprocessor extension to the conventional file system interface. Technical Report PCS-TR94-230, Dept. of Computer Science, Dartmouth College, September 1994.

[23] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[24] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[25] Rolf Rabenseifner and Alice E. Koniges. The Effective I/O Bandwidth Benchmark (b_eff_io).

[26] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.

[27] A. Ching A. Choudhary W. Liao R. Ross and W. Gropp. Noncontiguous I/O through PVFS, September 2002.

[28] R. Ross. Reactive Scheduling for Parallel I/O Systems, 2000.

[29] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters, 2002.

[30] Horst D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[31] Spec. `http://www.spec.org`.

[32] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, 1999.

[33] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.

[34] The ASCI I/O Stress Benchmark Codes. `http://www.llnl.gov/asci/purple/benchmarks/limited/ior/` .

[35] Version January Asci. Flash user's guide.

[36] Randolph Y. Wang and Thomas E. Anderson. xFS: A wide area mass storage file system. In *Workshop on Workstation Operating Systems*, pages 71–78, 1993.